“A RL Development Environment for Learning Moore Machine Network Polcies from Deep Recurrent Policies"

Nicholas W. Renninger*

I. Introduction

As an aerospace engineer, my interest in new, amazing machine learning technology like deep learning is somewhat tempered by the lack of explainability of Deep Neural Networks (DNNs), especially as pertaining to the solution of sequential decision-making problems. This means that while we have developed extensive methodologies in machine learning, motion planning, and formal methods for approaching humanity’s desire for autonomous systems, we currently lack robust ways to provide performance guarantees for autonomous, safety-critical systems. We are in dire need of more principled and guaranteed design methodologies than many "black-box" methods in use today if we want to realize trusted autonomy for safety-critical systems in our society.

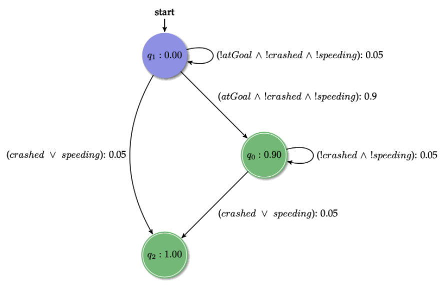

start
$q_1 : 0.00$
$(!atGoal \wedge !crashed \wedge !speeding)$: 0.05
$(atGoal \wedge !crashed \wedge !speeding)$: 0.9
$(crashed \vee speeding)$: 0.05
$q_0 : 0.90$
$(!crashed \wedge !speeding)$: 0.05
$(crashed \vee speeding)$: 0.05
$q_2 : 1.00$

Fig. 1 An example of a probabilistic deterministic finite automaton (PDFA) specification for one of my toy autonomous driving scenarios.

To bring us closer to provably safe, interpretable decision-making algorithms for autonomous systems, one approach I am investigating in my research is using machine learning to learn high-level human goals from demonstrations in a way that is amenable to formal control synthesis. Autonomous vehicles exemplify an industry that currently faces many challenges in the field of decision making and control for a safety-critical system. In his annual review in 2018, Schwarting outlined the progress made in the Verification and Synthesis of decision-making and motion planning algorithms [1]. Black box methods (i.e. DNN representing policy / value networks in Deep Reinforcement Learning) are struggling to be verified [1], which makes their use questionable in such a safety-critical environment. It was particularly noted that while formal control synthesis was of great interest, it does not see wide-spread use cause due to its expensive and the due to the difficulty in formally specifying desired system behavior.

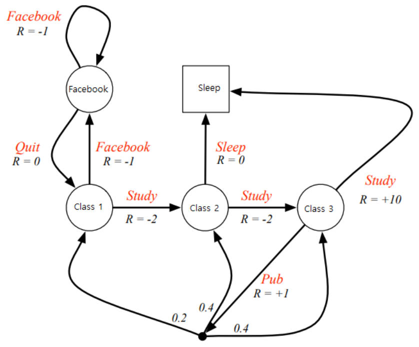


Fig. 2 An example of a Markov Decision Process (MDP) for which we might want to learn a specification. (source)

More specifically, my current research consists of first using probabilistic state-machine learning algorithms (e.g. ALERGIA, MDI [2], or RTI(+) [3]) to learn a probabilistic automaton model representing a human demonstrator’s specification for the autonomous system’s task. This learning process is based on observed symbol (a symbol is a label for a set of states) sequences (traces) representing system behavior. See Fig. 1 for an example of such a specification. These probabilistic state machines allow are generative in nature, and when sampled from produce traces in the "language" of the probabilistic automaton. This format for the learned specification is more useful than other generative language model (i.e. HMMs or Neural Networks) in one sense, as we can use tools from formal methods to create a controller for the autonomous system we got demonstrations of. Basically, if we can learn the specification for the demonstrator’s goals as a state machine, we can compose this specification with a model for the autonomous system’s dynamics (e.g. transition system or (PO)MDP – see Fig. 2) to obtain a correct-by-construction policy to control the system model to follow the learned specification.

*Graduate Student, Ann & H. J. Smead Department of Aerospace Engineering, 3775 Discovery Drive Boulder, CO 80303

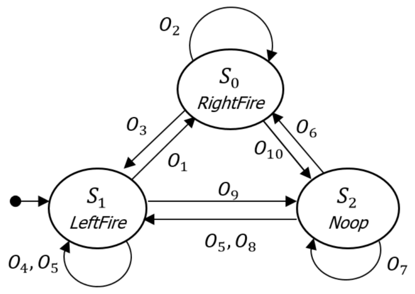


Fig. 3 An example of a three-state moore machine representing an optimal policy for pong for, given a CNN feature extractor that can output observations O_i.

For this project, I would like to focus on tackling both the **specification learning and synthesis** aspects of the project. The whole goal of learning a specification as an automaton is so we can generate a safe controller for the system from a model of the system and user demonstrations. What is instead all you had was a system to interact with and you wanted to learn an explainable, verifiable controller for the system? Thus, I think it would be very interesting to investigate deep reinforcement learning, as this is the exact sort of machine learning paradigm that would solve such a scenario. Using deep RL but in such a way as to learn a state-machine representation of the eventual policy is of interest to researchers working on explainable AI (XAI) as well as researchers like me working on formal control synthesis for uncertain sequential decision-making systems.

II. Problem Overview

Thus, in this project I am studying how RNN language models used as policy representations in a deep reinforcement learning setting fair as targets for state machine extraction. There is not much work in this specific field at all, but our approach would likely be based off of current work to extract state machines from an RNN representing the policy in the case of either model-based POMDP policy iteration [4] [5] or model-free reinforcement learning [6].

In this case, I will be interested in implementing the work done by Koul [6] (in the stable-baselines RL ecosystem), as this was the original work on extracting a moore-machine from an RNN encoding an RL agent's policy. This high-level approach is really interesting:

1) Use a baseline RL algorithm (e.g. DDPG, A3C, PPO) to train a CNN-LSTM policy for an atari game given as an RL environment. This would involve designing a good CNN and LSTM architecture that trains well for the given environment.
2) Next, we build the state machine *into* the policy network by quantizing both the feature space of the LSTM input and the hidden state space of the LSTM into a small, finite number of discrete states. This is accomplished by training two autoencoders – one for the feature quantization and one for the hidden state quantization. These autoencoders are trained by feeding features and traces from the replay buffer through these compression networks and minimizing the reconstruction loss given only binary or tenrary latent states.
3) Then, these autoencoder pairs (called quantized bottleneck networks (QBNs)) are inserted back into the original policy network before and after the recurrent layers. As the quantization networks have already been trained, only fine tuning of this new policy network should be necessary with more RL episodic training.
4) You now have what they call a moore-machine network, which is a neural network moore machine that represents the agent's policy. This is the state machine that we have been looking for!

As our learning target is now a type of moore machine, let's define one. A moore machine is a finite, deterministic input / output state machine. For the Atari game pong, we can see a classical moore machine controller for it in Figure 3. A moore machine network works by taking a continuous observation of the world, and using its first QBN to encode this into some discrete state (think of the latent state of the QBN as being a binary / ternary encoding of the input). This discrete state is decoded and re-encoded by the rest of the LSTM and second QBN into a discrete hidden state - the state of the moore machine. This state is then decoded into an action, and then from the current hidden state the machine receives its next observation and the process continues. Hopefully it is clear now that the network in this case is being forced to act like a discrete state machine, despite being originally a purely continuous transformations of observations into actions.

So our major goal for this project is to pick a reinforcement learning environment studied by Koul et. al. and then try to learn a CNN-LSTM policy, two QBNs (one for observation features and one for hidden state), and a moore machine network policy to control. This involves migrating the lightweight research code for the Koul et. al. paper and integrating the custom models and training procedures into the stable-baselines, and thus for TF1.15. Having this implementation available will allow for the eventual extraction and learning of a finite state moore machine controller from the moore machine network.

Another large goal of this entire project was to build a reproducible environment for studying various deep reinforcement learning techniques. As such, all of this work was built with a custom docker environment that serves jupyter and tensorboard servers over the host's network. All of this is contained in my github repo. This container and its jupyter / tb servers can also be deployed on a remote cluster and then run through ssh on the remote machine - indeed this is how this work was all developed. The github repo contains comprehensive documentation on this project, including installation and usage. Also, all of the results here are contained in a very large, detailed jupyter notebook that walks through all of the steps to solve this problem in explicit detail. Along with this, all of the implementations I did are fully integrated with stable-baselines (the largest and most comprehensive RL library) and have been migrated from PyTorch to TF1.15 (tensor library underpinning stable-baselines), which should allow the work done Koul et. al. to be more widely applied to the vast RL ecosystem around stable-baselines. Thus, the advancement made here is largely on the implementation side. Check out the repo to get all of the documentation for the environment.

III. Methodology

Here is a high-level overview of the steps I took to learn a moore machine network (MMN) controller, following Koul et. al. [6], with the high-level steps for the process shown in Figure 4:

1. Learn an feature_extractor-rnn_policy for a RL environment using a standard RL algorithm capable of learning with a recurrent policy, PPO2. PPO is a policy gradient method that essentially allows for monotonic policy improvements like in TRPO [7][8]. It develops a "surrogate" objective function $L_t^{CLIP+VF+S}(\theta)$ that can be approximately mazimized each step [8]:
   $L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t)\right]$
   Where $L_t^{CLIP}(\theta)$ is the clipped surrogate objective that is the basis for the whole procedure, where it lower bounds another constrained policy objective, the conservative policy iteration objective [8]. $c_1 L_t^{VF}(\theta)$ is the scaled value estimate produced by the actor-critic portion of the policy network, and $S\left[\pi_\theta\right](s_t)$ is an entropy bonus to encourage the production of explorative actions (we don't want too simple of a policy - likely not a good one) [8].
   Here the feature extraction network is known as `F_ExtractNet` and the RNN policy that takes these features and produces the next action is known as `RNN_Policy`. *If your environment already has simple, discrete observations, you will not need `F_ExtractNet` and can directly feed the observation into the `RNN_Policy`.* Here, we use the stable-baselines implementation of PPO2 with our custom policy networks to train the CNN-LSTM agent [9].
2. Generate "Bottleneck Data". This is where you simulate many trajectories in the RL environment, recording the observations and the actions taken by the `RNN_Policy`. This is for training the "quantized bottleneck neural networks" (`QBNs`) next.
3. Learn `QBNs`, which are essentially applied autoencoders (AE), to quantize (discretize):
   - the observations of the environmental feature extractor:
     - CNN if using an agent that observes video of the environment.
     - MLP if getting non-image state observations This is called `b_f` in the paper and `OX` in the mnn code.
   - the hidden state of the `RNN_Policy`. This is called `b_h` in the paper and `BHX` in the mnn code

This is done by taking the bottleneck data and using to train the QBNs as if they were autoencoders,

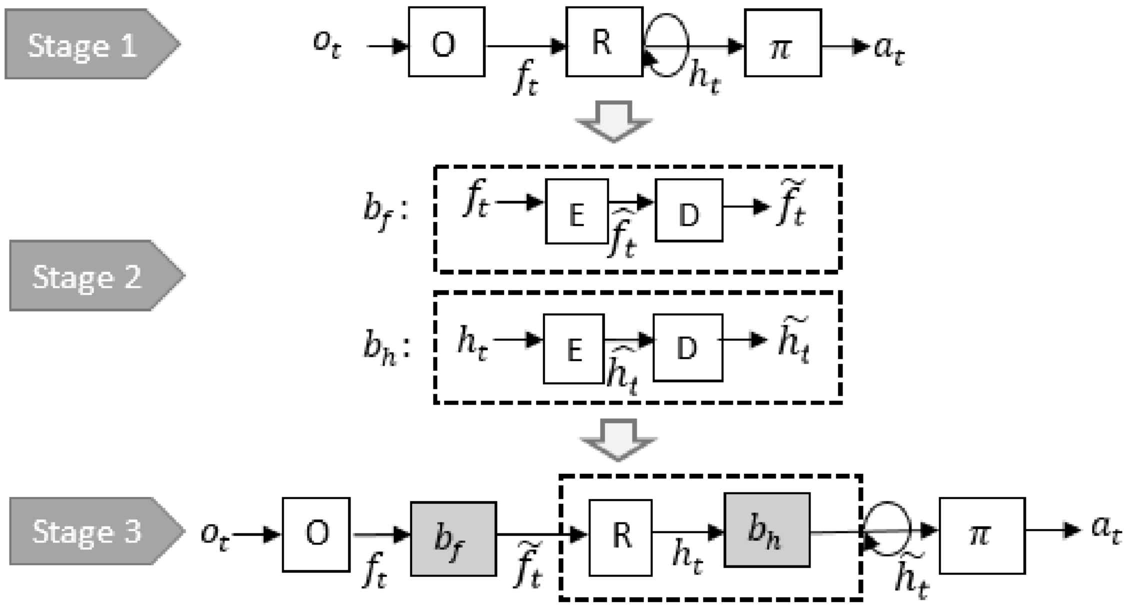


Fig. 4 The conceptual building steps for the moore machine network [6]. Note that the bottlenecks (b_f and b_h) are trained separately and *then* inserted in the regular policy ($o_t - O - R - h_t - \pi - a_t$).

using a reconstructive MSE loss on the input and reconstructed input from the QBN. One thing worth noting here is that in order for the QBN's quantized state comes from using a ternary activation on the inputs to the latent layer [6]. This way, each hidden neuron in the latent layer of each QBN has a value of either -1, 0, or 1. In this way, we gain a compressed AND quantized representation of the input at the latent layer of the network.

This ternary activation ϕ is given by $\phi(x) = 1.5 \tanh(x) + 0.5 \tanh(-3x)$ [6]. As suggested by Koul et. al., we must find a way to estimate the gradient of this activation, as it is non-differentiable. Koul et. al., referencing work by others in quantization of network representations, treats the function as the identity during backprop [6].

4. Inserting the trained `OX` QBN *before* the feature extractor and the trained `BHX` QBN *after* the RNN unit in the feature_extractor-rnn_policy network to create what is now called the moore machine network (`MMN`) policy.
5. Fine-tune the `MMN` policy by re-running the rl algorithm using the `MMN` policy as a starting point for RL interactions. *Importantly, for training stability the `MMN` is fine-tuned to match the softmax action distribution of the original `RNN_Policy`, not the argmax -> optimize with a categorical cross-entropy loss between the RNN and `MMN` output softmax layers.*

Table 1 Network Architecture for the CNN-LSTM Policy Trained with PPO2. Input flows sequentially downwards through layers unless otherwise noted.

Layer Name	Layer Parameters
conv1	n_filt=32, filt_size=8, stride=4, bias=True, act=ReLU
conv1	n_filt=32, filt_size=8, stride=4, bias=True, act=ReLU
conv1	n_filt=32, filt_size=8, stride=4, bias=True, act=ReLU
fc1	n_hidden=512, bias=True, act=ReLU
LSTM	n_hidden_cells=256, act=Tanh
value_est_fc	n_hidden=1, act=Linear (linked to LSTM out)
actions_dist_fc	n_hidden=n_actions=6, act=softmax (linked to LSTM out)
Q_est_fc	n_hidden=n_actions=6, act=Linear (linked to LSTM out)

Table 2 Hyperparameters for the CNN-LSTM Policy Trained with PPO2

Hyperparameter Name	Value
n_parallel_envs	16
cliprange	linear_decay(0.1)
ent_coef	0.01
γ	0.99
λ	0.95
learning rate	linear_decay(4e-4)
optimizer	Adam() default betas
n_steps	128
n_timesteps	7,000,000
nminibatches	8
noptepochs	4

Table 3 Network Architecture for the Observation Feature QBN. Input flows sequentially downwards through layers unless otherwise noted.

Layer Name	Layer Parameters
fc1-enc	n_hidden=fc3_out.flatten()=512, act=Tanh
fc2-enc	n_hidden=8*n_latent_states=800, act=Tanh
latent-enc	n_hidden=100, act=ternary_tanh
fc1-dec	n_hidden=8*n_latent_states=800, act=Tanh
fc2-dec	n_hidden=512, act=ReLU6

Table 4 Hyperparameters for the Observation Feature QBN

Hyperparameter Name	Value
learning rate	linear_decay(4e-4)
optimizer	Adam() default betas
global norm gradient clip	5.0
loss	MSE
batch_size	32
epochs	600
n_training_ex	5000

Table 5 Network Architecture for the Hidden State Feature QBN. Input flows sequentially downwards through layers unless otherwise noted.

Layer Name	Layer Parameters
fc1-enc	n_hidden=fc3_out.flatten()=256, act=Tanh
fc2-enc	n_hidden=8*n_latent_states=640, act=Tanh
fc3-enc	n_hidden=4*n_latent_states=320, act=Tanh
latent-enc	n_hidden=80, act=ternary_tanh
fc1-dec	n_hidden=4*n_latent_states=320, act=Tanh
fc2-dec	n_hidden=8*n_latent_states=640, act=Tanh
fc3-dec	n_hidden=512, act=Tanh

Table 6 Hyperparameters for the Hidden State Feature QBN

Hyperparameter Name	Value
learning rate	linear_decay(4e-4)
optimizer	Adam() default betas
global norm gradient clip	5.0
loss	MSE
batch_size	32
epochs	600
n_training_ex	5000

IV. Results

A. Training the CNN-LSTM Agent Using PPO2

In Tables 1 and 2 we defined the CNN-LSTM agent and the PPO2 hyperparameters used to train the agent. Figure 6 shows the "main" policy gradient loss evolution with environment step, while Figures 7 - 9 show how the individual components of this overall loss evolve.

The agents episodic reward is shown in Figure 10, where we can see that at the end of the 7M training steps, the agent is consistently getting the maximum reward for the environment. The vectorized training environment for the CNN-LSTM can be seen in Figure 5.

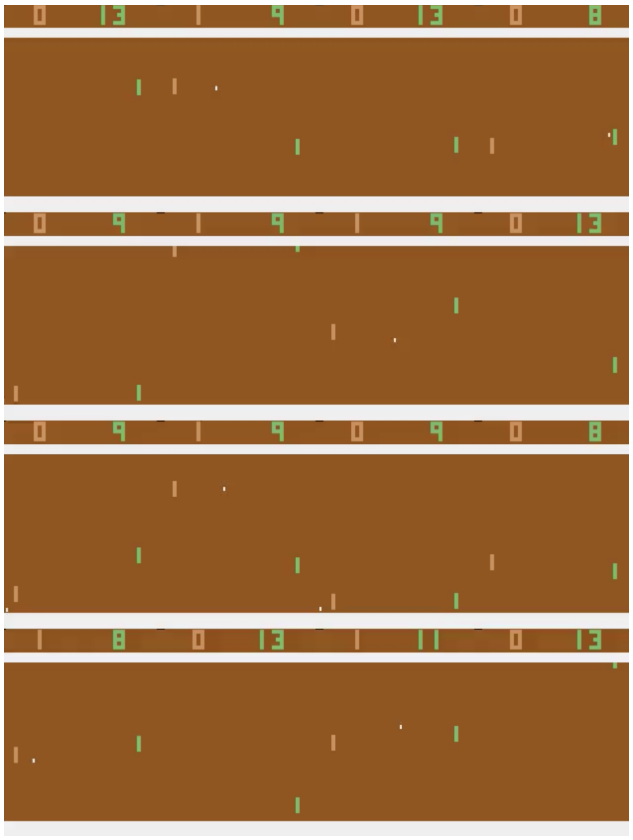


Fig. 5 A figure showing the vectorized environment the PPO2 agent trained in. As PPO2 works with rollouts of the policy and not active interaction, rollouts are easily parallelized and greatly aid training speed. Training in a parallelized environment took many hours fewer than in scalar environment.

B. Training the Quantized Bottleneck Networks

The observation QBN was trained using the parameters found in Tables 3 & 4. We can see the evolution of the reconstruction MSE loss on the sampled observation features from the CNN of the CNN-LSTM policy with training epoch in Figure 12.

The hidden state QBN was trained using the parameters found in Tables 5 & 6. We can see the evolution of the reconstruction MSE loss on the sampled hidden states of the LSTM module in the CNN-LSTM policy with training epoch in Figure 11.

C. Evaluation of Trained Agents

Once the agent and the QBNs were trained, we could insert the QBNs into the CNN-LSTM policy to create the Moore Machine Network. Again, it is called a moore machine network because the ternary-quantized latent space encoding of the QBNs makes the whole QBN-CNN-LSTM have discrete observation states that cause the whole model to transition to a new discrete hidden state and emit an action. As can be seen in Table 7, fine-tuning of the MMN was not entirely necessary, as the MMN still performed similarly, albeit noticeably worse than the base CNN-LSTM. Shown in Figure 13, we can see how the learned agents stack up to each other and how they qualitatively differ.

Table 7 Average Non-discounted Reward Over Monte-Carlo Rollouts

Original CNN-LSTM Agent	MMN Agent
20.3 ± 0.2	18.90 ± 1.14

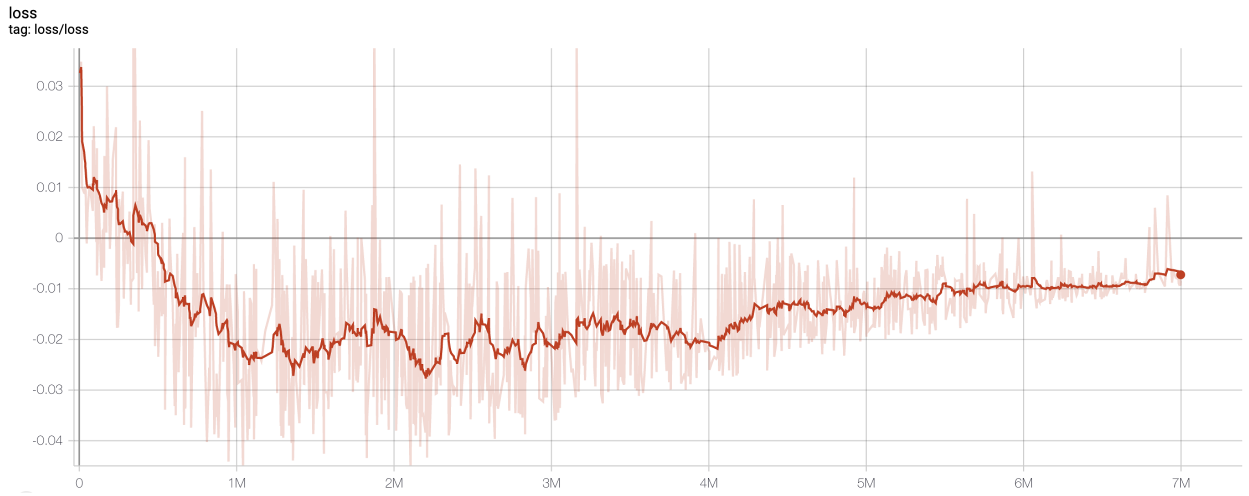


Fig. 6 The full PPO2 loss term (objective function) per environmental step, $L_t^{CLIP+VF+S}(\theta)$, of the overall policy gradient loss used by PPO2, with the final network and training settings. We can see that just as with most RL algorithms, even with extremely large numbers of samples the loss does not go to zero, especially as the objective function for PPO2 is only approximately optimized during training [8].

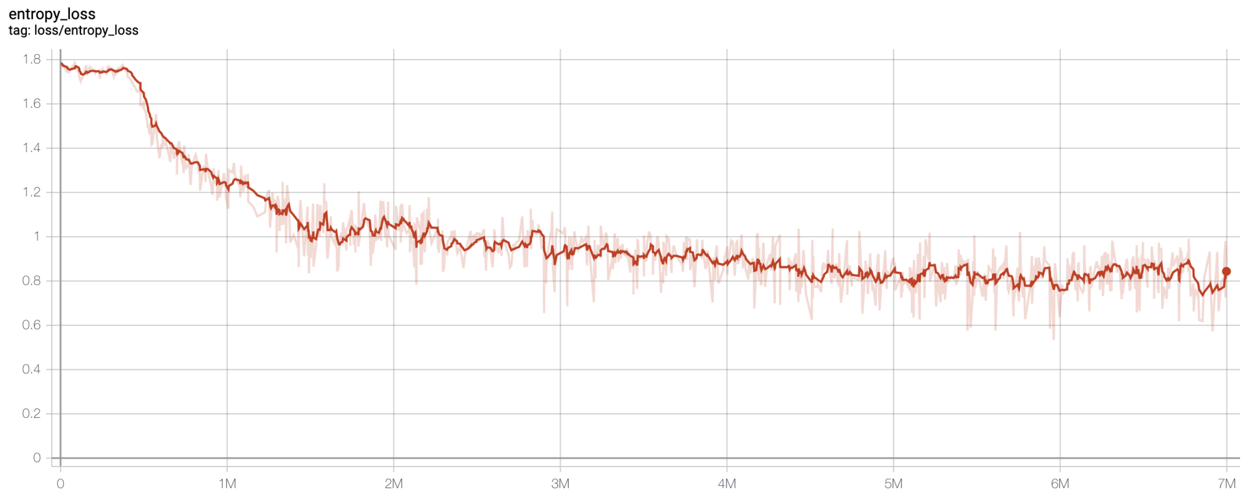


Fig. 7 The entropy term per environment step, $S[\pi_\theta]$, of the overall policy gradient loss used by PPO2, with the final network and training settings. Notice that the entropy, which should be maximized as an objective in PPO2 [8], actually decreases with each episode. I suspect the agent began trading off policy entropy for perhaps better surrogate objective ($L_t^{CLIP}(\theta)$).

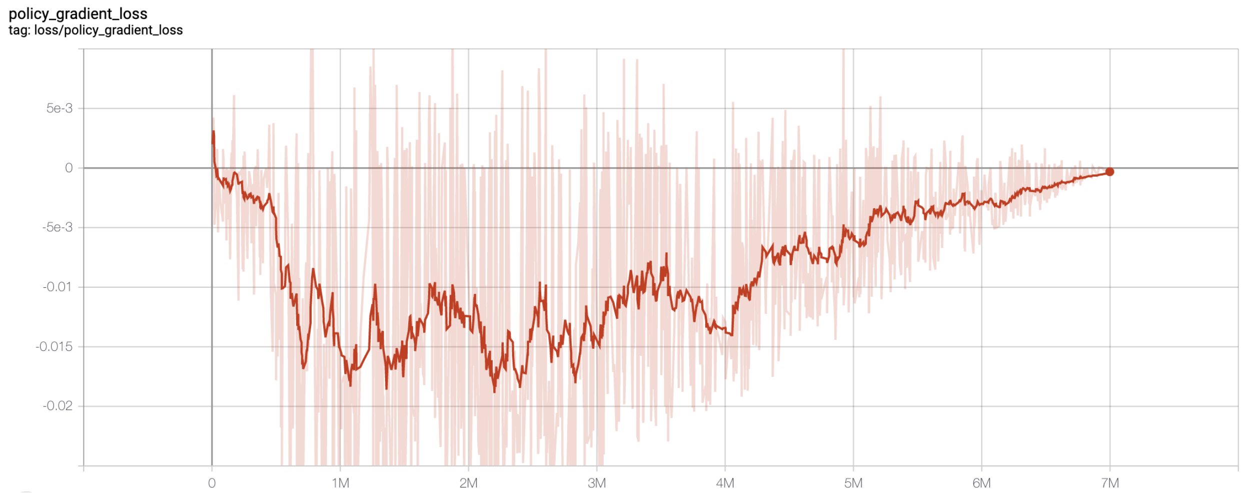


Fig. 8 The surrogate PPO2 loss term (the objective function based on a trust region [8]) per environmental step, $L_t^{CLIP+VF+S}(\theta)$, of the overall policy gradient loss used by PPO2, with the final network and training settings. The agent needed all 7M steps to reduce this loss term, here based on the "probability ratio clipping" surrogate loss [8], to 0 which it impressively did.

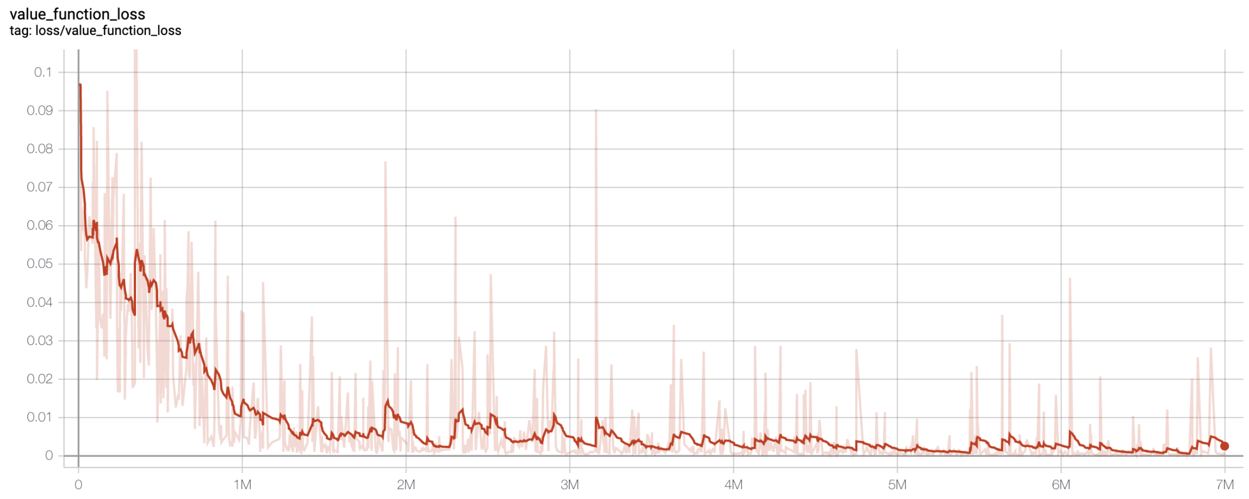


Fig. 9 The value PPO2 loss term per environmental step, $L_t^{VF}(\theta)$, of the overall policy gradient loss used by PPO2, with the final network and training settings. This loss, which is based on the actor-critic architecture of the output layers, was nicely driven to 0 by the end of the training process.

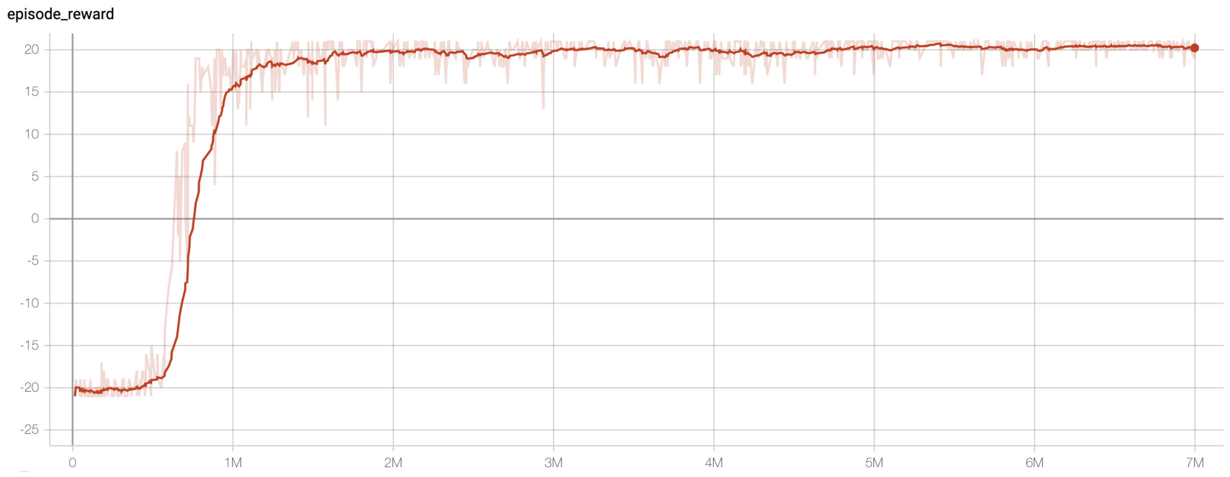


Fig. 10 The PPO2 agent's episodic reward. While it may appear that we could have finished training much earlier than at 7M steps, based on the loss curves shown in the other plots, the agent was still stabilizing and internally improving. Qualitatively, my experience was that the agent needed significant time after first reaching the maximum episodic reward (21) to be reliable and smooth when visualized. A good example of how reward signals are rarely capable of encoding the fully desired task specification.

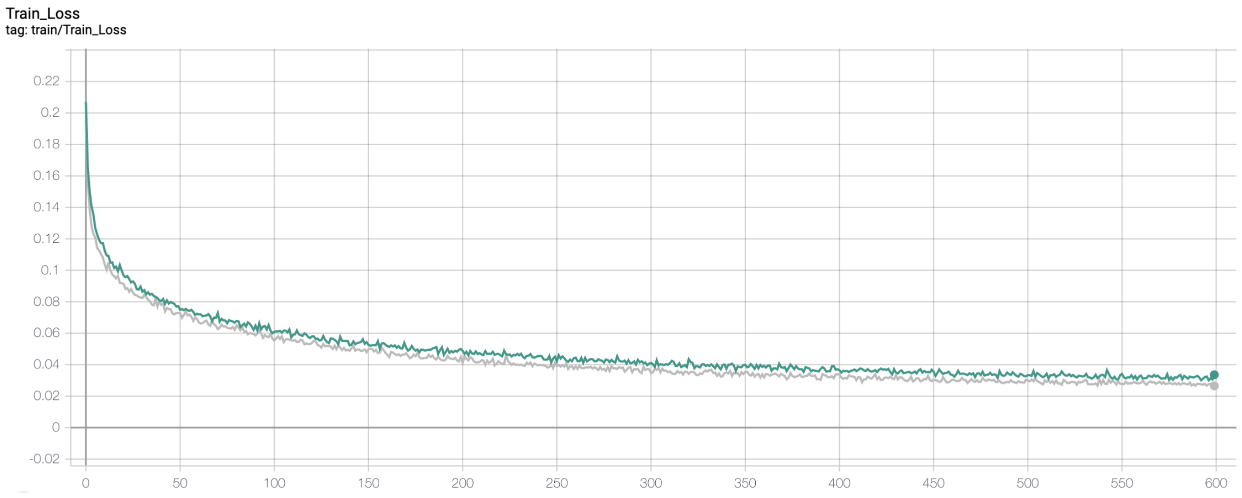


Fig. 11 The training and test losses vs. epoch for the final QBN model used to quantize and compress the hidden state output of the LSTM module of the learned Atari agent's policy.

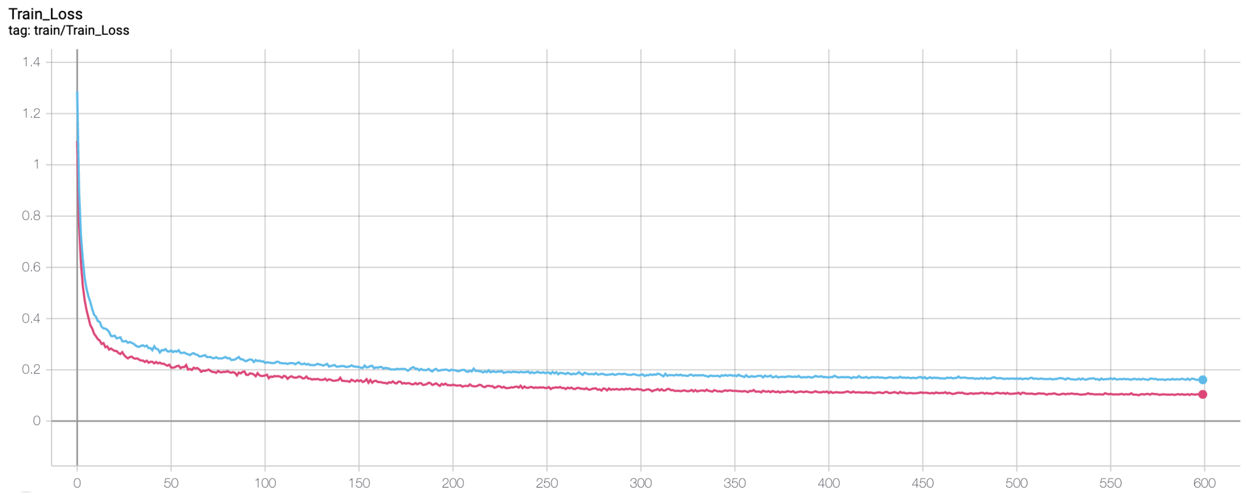
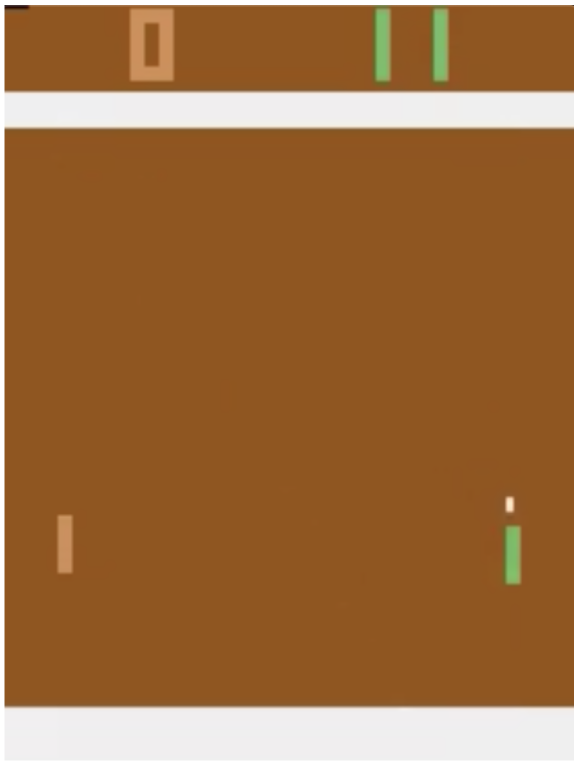
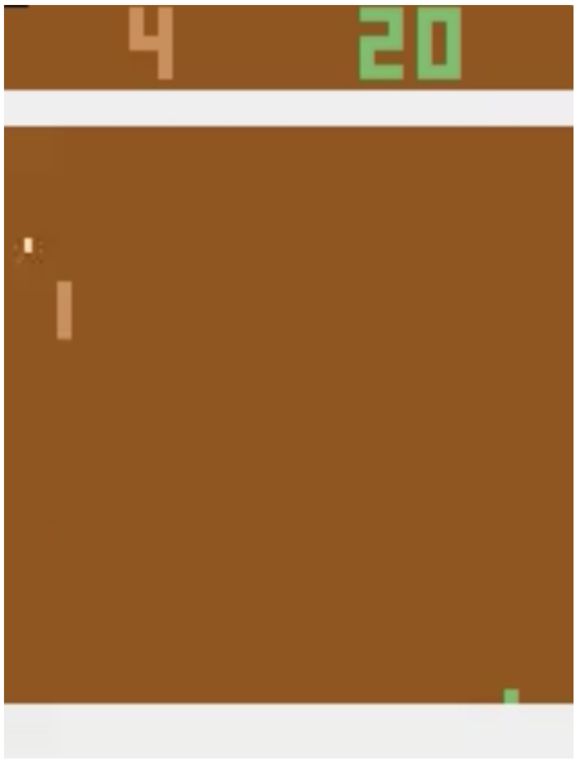


Fig. 12 The training and test losses vs. epoch for the final QBN model used to quantize and compress the flattened convolutional feature output of the CNN feature extractor module of the learned Atari agent's policy. I had a lot of trouble getting this the same QBN architecture used in the Koul et. al. paper [6] to converge to a near-zero loss, but I was also using the original atari CNN architecture [10] instead of the that in the Koul paper - a more complex CNN than the one use by Koul et. al.



(a) This is showing the fully PPO2 trained CNN-LSTM agent, under-the-hood transformed to work with a single environment, showcasing its superhuman abilities. Believe it or not, the agent actually returned the ball right after this step. Also note the score, 11-0, anecdotal confirmation of the agent's dominance and strong learning performance.



(b) This is showing the fully "trained" MMN agent. Both the observation feature QBN and hidden state QBN have been fully trained, as in Figures 12 & 11 respectively, and inserted into the original CNN-LSTM policy network as shown in Figure 4. Here, the MMN agent winning the game, but it obvious that it lost a little bit of performance compared to the original policy (look at the score). Also, if you check the github repo to see gifs of the MMN agent, where it is certainly clear that while still performant, the quantization has made it appear visually less "smooth".

Fig. 13 **Visualizations of the CNN-LSTM and MMN Pong Agents in Different Environments**

V. Conclusion

As we saw in the Results section, the CNN-LSTM and MMN agent both performed very well and were eventually found to be quite trainable. There is also the positive result that Koul et. al.'s work turned out to be quite reproduceable, a rarity in the world of AI research. The idea of QBN insertion is a potentially very powerful one, but needs more investigation. I am immediately thinking of using it along with power deep learning based feature extractors to help symbolize continuous data I have into symbols that I can use for specification learning. THe only problem with this idea is that this quantization step was not really well compared with any other way of doing feature compression and quantization. However, as with most things in deep learning, it is likely that deep feature representations will be more powerful and easier to construct than hand-designed features, so overall the results look promising.

I think a nice extension of this work would be to apply this moore machine creation to imitation learning paradigms, so you can learn an explainable controller from just system demonstrations. Another interesting thing to try would be to try to learn other types of state-machines using this method, as I don't typically use moore machines in my research.

References

[1] Schwarting, W., Alonso-Mora, J., and Rus, D., "Planning and Decision-Making for Autonomous Vehicles," *Annual Review of Control, Robotics, and Autonomous Systems*, Vol. 1, No. 1, 2018, pp. 187–210. doi:10.1146/annurev-control-060117-105157, URL https://doi.org/10.1146/annurev-control-060117-105157.

[2] de la Higuera, C., *State Merging Algorithms*, 2004, Cambridge University Press, New York, NY, USA, 2013, Chap. 16.3, pp. 333–339. doi:10.1017/CBO9781139194655.

[3] Verwer, S., Eyraud, R., and Higuera, C., "PAutomaC: A Probabilistic Automata and Hidden Markov Models Learning Competition," *Mach. Learn.*, Vol. 96, No. 1-2, 2014, pp. 129–154. doi:10.1007/s10994-013-5409-9, URL https://doi.org/10.1007/s10994-013-5409-9.

[4] Carr, S., Jansen, N., Wimmer, R., Serban, A., Becker, B., and Topcu, U., "Counterexample-guided strategy improvement for POMDPs using recurrent neural networks," *IJCAI International Joint Conference on Artificial Intelligence*, Vol. 2019-Augus, 2019, pp. 5532–5539. doi:10.24963/ijcai.2019/768.

[5] Carr, S., Jansen, N., and Topcu, U., "Verifiable RNN-Based Policies for POMDPs Under Temporal Logic Constraints," Tech. rep., 2020. URL https://arxiv.org/pdf/2002.05615.pdf.

[6] Koul, A., Fern, A., and Greydanus, S., "Learning finite state representations of recurrent policy networks," *7th International Conference on Learning Representations, ICLR 2019*, 2019, pp. 1–15.

[7] Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., and Moritz, P., "Trust Region Policy Optimization," *ICML*, 2015.

[8] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O., "Proximal Policy Optimization Algorithms," *ArXiv*, Vol. abs/1707.06347, 2017.

[9] Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y., "Stable Baselines," https://github.com/hill-a/stable-baselines, 2018.

[10] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D., "Human-level control through deep reinforcement learning," *Nature*, Vol. 518, No. 7540, 2015, pp. 529–533. doi:10.1038/nature14236, URL https://doi.org/10.1038/nature14236.